1978

Matching Tree Patterns

Christoph M. Hoffmann
Purdue University, cmh@cs.purdue.edu

Report Number:
78-291

MATCHING TREE PATTERNS

Christoph M. Hoffmann
Computer Science Department
Purdue University
West Lafayette, Indiana 47907

CSD-TR 291

December 1978

MATCHING TREE PATTERNS

Christoph M. Hoffmann
Computer Science Department
Purdue University

Summary

We investigate the problem of matching tree patterns which contain variables which can match arbitrary subtrees. An efficient solution to this problem provides a direct implementation of solutions to a number of problems, many of which are related to issues of implementing programming languages. Such direct implementations can be faithful to a precise underlying mathematical model of the problem.

We show that a general matching algorithm has to tackle an exponential explosion of different partial matches possible in some tree in the case of certain patterns. We isolate an easily recognized subclass of patterns, for which this exponential growth does not take place, and develop efficient algorithms for this class.

We also exhibit similarities between matching tree patterns and matching string patterns, pointing out the factors which cause tree patterns to behave in a more complicated manner, and discuss similarities of our algorithms to previously reported algorithms for matching strings efficiently.

1. Introduction

We consider matching algorithms for tree patterns. Matching of this kind occure in a number of applications, and especially in the implementation of Subtree Replacement Systems (SRS). In a SRS a tree is transformed by successively replacing subtrees with new subtrees, until no further replacements are possible. The replacement transformations are expressed as a set of rules, that is, pairs of tree schemata: Given that a subtree matches an instance of a rule's lefthand side, it is replaced with the corresponding instance of the righthand side.

For a theoretical study of SRS see, for instance, [Ros 73] and [O'D 77]. Among the problems which SRS model naturally we have

- Automatic interpreter generation [H&O 79]: The generator program processes essentially a set of replacement rules precisely defining the semantics of the programming language to be interpreted. From it, tables driving a standard SRS algorithm are produced, which then implements an interpreter for the defined language.

- Direct implementation of abstract data types, e.g. [GHM 76]: The axioms defining the operations on the type are viewed as SRS rules. From these, an interpreter may be generated as above, thus faithfully interpreting the operations without manipulating arbitrary concrete representation maps.

- Certain code optimization techniques [Sta 77]: Intermediate code is represented as a forest of (attributed) abstract syntax trees. Optimizations, such as elimination of redundant operations, or constant propagation, are expressed as subtree replacement rules. Sometimes, parts of code generation may be accomplished in this way too.

- Symbolic computation [Col 71]: Algebraic terms are represented as trees. Transformations which formalize operations such as differentiation and certain algebraic simplifications are used to derive result terms.

In all of the above cases, an implementation based on efficient algorithms for matching tree patterns becomes a desirable alternative because of its faithfulness to the underlying equational model. Furthermore, as discussed in [UKO 79], such implementations may be generated mechanically.

There are other areas in which efficient tree matching algorithms have relevance. First order unification, e.g. [Rob 65, Bax 76, P&W 76], deduces equality of two terms through substitution, and may be viewed as matching two tree patterns against each other. The exact relation of the problems addressed in this paper and unification is clarified in Section 2.

There has been considerable work on deducing the equivalence of two terms from a set of axioms, e.g. [N&O 78, DSS 78, Sho 78]. The problem may be formulated within the SRS framework, i.e. do two terms reduce to equal normal forms. However, the cited works have not approached the problem from this perspective, and do not utilize pattern matching techniques. An exception to this is [K&B 70], which gives methods for testing term equivalence by reducing them. However, no algorithms for tree pattern matching are given.

Matching tree schemata generalizes matching string patterns when considering strings as nonbranching trees, that is, trees in which each interior node has exactly one descendant. We discuss the relation of our tools to the methods invented for fast pattern matching of strings, especially to the work of [A&C 75] and [KMP 77]. For this we assume some familiarity with the basic ideas behind the algorithms. We have not found ways of generalizing the approach of [R&W 77].

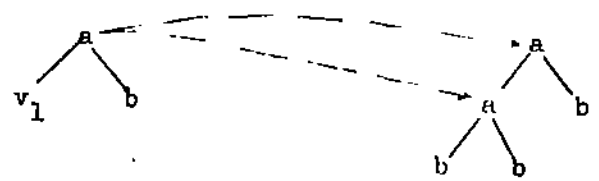
Section 2 of the paper formally defines the matching problem we consider, and derives general properties which give insight into the intrinsic difficulties of the problem and their sources. Section 3 isolates a subproblem by suitably restricting the form of tree patterns, such that the difficulties of the general problem can be avoided, and derives useful special properties of the subproblem which allow designing good algorithms for its solution. These algorithms are developed in detail in Sections 4 and 5. Section 6 summarizes our results.

2. Tree Patterns and Match Sets

We are given a finite ranked alphabet Σ of function symbols, including constants. We also have at our disposal a countable set V of variables. S denotes the usual set of Σ-terms, S_V the set of $\Sigma \cup V$-terms. We consider all terms to be trees.

A tree pattern is any term in S_V. Intuitively, a pattern t with variables $v_1, \ldots, v_k$ matches in a tree t' in S, with each of the v_i matching arbitrary subtrees in t'. More formally, t matches t' at node p, if there are trees $t_1, \ldots, t_k$ in S such that substituting t_i for each occurrence of v_i in t, $1 \leq i \leq k$, we obtain a tree t'' equal to the subtree of t' rooted at p. The size of a tree is the total number of subtrees in it, i.e., the total number of nodes in the tree. The height of a tree is the length of a longest path from the root to a leaf of the tree.

Example 2.1 The pattern $a(v_1, b)$ matches in the tree a(a(b, b), b) in two ways, with v_1 matching a(b, b) and b, respectively. The dashed lines indicate pattern alignments.



We can now state the matching problem: Given a finite set F of patterns $t_1, \ldots, t_k$, $F \subset S_V$, such that in each t_i no variable symbol is repeated, locate in a tree t in S all possible matches of the t_i.

The restriction on variable repetition is motivated by theoretical problems which arise when repeated variables are permitted in the specification of SRS rule lefthand sides. See [O'D 77], Section VII for further details.

As stated, the matching problem has the following relation to first order unification: First order unification matches two patterns t_1 and t_2 against each other, but aligned at their roots. We restrict patterns not to contain repeated variables, but match them anywhere in trees which do not themselves contain variables, rather than matching at the root alone.

There is a linear algorithm solving the unification problem [P&W 76]. Since we do not allow repeated variables, the algorithm can be simplified and adapted to solve the matching problem in $O(n \cdot m)$ time and $O(n+m)$ space, where n is the size of the subject tree, and m the sum of the pattern sizes. We refer to this adaptation as the "naive" algorithm, since it closely resembles the naive algorithm for string matching: Align the tree patterns to be matched in every possible way in the subject tree - there are $O(n)$ possibilities. For each alignment position, match the patterns by traversing them, say in preorder, matching corresponding tree nodes and checking for equal labels, except for pattern leaves which are labelled with a variable symbol. These traversals require $O(m)$ steps, since there are no repeated variables.

In many applications of tree pattern matching, especially in the implementation of SRS, the set of patterns remains fixed and is to be matched repeatedly in a number of subject trees. For such applications it is advantageous to preprocess the patterns if this can lead to faster matching algorithms. Specifically, consider the following.

From the set F of tree patterns, compute all possible sets of (partial) matches which can ever occur. There must be a finite number of those. Then, for each alphabet symbol, construct a table used in this way: Process the tree t in which to match the patterns in F from the leaves of t up, assigning to a node p in t a code representing the set of all (partial) matches at p. From the precomputed tables, this code for p can be assigned in constant time as function of p's label and the codes assigned to p's sons. (If p is a leaf,

only one code can ever be possible.) Whenever the match set represented by some code c contains a complete pattern t_i, then t_i has been matched at each node to which c is so assigned. The algorithm, hereafter referred to as Algorithm M, finds all occurrences of the patterns in F in O(n) time, where n is the number of nodes in t, given the precomputed tables, and is especially well-suited to the additional requirements of SRS implementations.

Given a forest F of tree patterns, a match set M is a set of (sub)trees in F such that there is some Σ-tree t with every t' in M matching t at the root, and such that every other (sub)tree in F which is not in M does not match t at the root. Visualize M as the set of all (partial) matches at the root of t.

Example 2.2 Given the pattern forest $F = \{a(b,v_1),\ a(a(v_2,v_3),b)\}$, the set $M = \{a(b,v_1),\ a(v_2,v_3),\ v_1,\ v_2,\ v_3\}$ is a match set because of $t = a(b,b)$, but the set $\{a(b,v_1),\ v_1\}$ is not a match set, since, for example, a match of $a(b,v_1)$ always implies a match of $a(v_2,v_3)$, at the same node.

Since we ruled out repeated variables in patterns, we collectively denote variable symbols by v, observing that different occurrences of v may match different subtrees.

To analyze match sets, we define three relations on tree patterns. A tree t in S_V is inconsistent with t' in S_V, $t \parallel t'$, if both t and t' cannot be matched at the same node in any tree in S, that is, no match set may contain both t and t'. For example, $a(b,v) \parallel a(c,v)$. Trees t and t' are independent, $t \sim t'$, if there are trees t_1, t_2 and t_3 in S such that t matches t_1 and t_3 at the root, but not t_2, whereas t' matches t_2 and t_3 at the root, but not t_1. For instance, $a(b,v) \sim a(v,c)$ because of the trees a(b,b), a(c,c) and a(b,c). Finally, t subsumes t', $t > t'$, if a match of t always implies a match of t' at the same node; e.g. $a(b,v) > a(v,v)$.

Given two trees t and t' in S_V, it is clear that exactly one of the above relations must hold for t,t' or t',t. Elementary properties of the relations

are summarized below without proof. Note that if no pattern contains any variables, then the only relation which holds between pattern (sub)trees is inconsistency, or equality.

Proposition 2.1 For distinct trees t_1, t_2, t_3 in S_V

(a) $t_1 > t_2$ and $t_2 > t_3$ implies $t_1 > t_3$ (transitivity)

(b) $t_1 \sim t_2$ iff $t_2 \sim t_1$ } (symmetry)

(c) $t_1 \| t_2$ iff $t_2 \| t_1$ }

(d) $t_1 \| t_2$ and $t_3 > t_2$ implies $t_1 \| t_3$ (propagation upwards)

(e) $t_1 \sim t_2$ and $t_2 > t_3$ implies $t_1 \sim t_3$ or $t_1 > t_3$ (propagation downward)

Given a pattern forest F, we wish to partition each match set M for F into a set M_0 of pairwise independent trees, and a set M_1, such that each tree in M_1 is subsumed by some tree in M_0. M_0 is called the base of M.

Proposition 2.2 Given a pattern forest F and a match set M for F, then there is a unique partition of M into sets M_0 and M_1 such that, for different t_1, t_2 in M_0 $t_1 \sim t_2$ is true, and for each t' in M_1 there exists a t in M_0 such that t subsumes t'.

Proof By transitivity, there is a unique set M_0 of trees t in M not subsumed by any other tree in M. Since distinct trees in M_0 cannot be inconsistent nor subsume one another, they must be independent. By definition of M_0, $M_1 = M - M_0$ has also the required properties. Assume now that M can be partitioned differently into sets M_0' and M_1' which also satisfy the proposition. Then, by definition of M_0, $M_1 \subseteq M_1'$ and $M_0' \subseteq M_0$. If there is some $t' \in M_1' - M_1$, then, by assumption, there is some t in M_0' and hence in M_0 such that $t > t'$. But also $t' \in M_0$, hence $t \sim t'$ as well, which is a contradiction. Hence $M_0' = M_0$ and $M_1' = M_1$. ∎

Observe that different match sets must have different base sets. We could, therefore, represent match sets by their base sets.

Given a pattern forest F, construct the <u>independence graph</u> G_I of F as follows: The vertices of G_I are distinct (sub)trees in F. There is an (undirected) edge between vertices t and t' iff t and t' are independent.

<u>Example 2.3</u> Consider a pattern forest formed by three trees where $t_1 = a(b(b(v)),v)$, $t_2 = a(b(v),b(v))$, $t_3 = a(v,b(b(v)))$. The distinct subtrees in this forest are $t_4 = b(b(v))$, $t_5 = b(v)$, and $t_6 = v$. Since the trees t_1, t_2, t_3 are pairwise independent, whereas no other tree pairs are, the independence graph G_I of this forest is

with a connected component t_1, t_2, t_3 and three isolated points.

<u>Theorem 2.3</u> The number of possible match sets of a pattern forest F cannot exceed the number of cliques in the independence graph of F, counting all subcliques, including the trivial ones.

<u>Proof</u> The base sets of distinct match sets must be different (sub)tree sets of F. Since the trees in each base set are pairwise independent, they must form a clique in G_I. ■

The upper bound provided by Theorem 2.3 need not be attained, as shown by the forest of Example 2.3: Note that a match set with base $\{t_1, t_3\}$ cannot occur, since matching both t_1 and t_3 at the same node implies matching t_2 as well, but neither t_1 nor t_3 subsume t_2. We would have to analyze deeper structural properties to arrive at exact bounds, which is beyond the scope of this paper.

The graphs G_I could be such that for certain pattern forests the number of cliques grows exponentially with the number of (sub)trees in F, hence with

the size of the forest. In such cases, the number of distinct match sets may also grow exponentially:

Theorem 2.4 There are pattern forests for which the number of distinct possible match sets grows exponentially with the size of the forest, i.e. with the number of (sub)trees in F.

Proof We define a class of balanced binary trees t^i_j, $0 \le i$, $0 \le j \le 2^i$, of height i, such that all interior nodes are labelled a, and in t^i_j all leaves are labelled with the variable symbol v, except the j-th leaf from the left which is labelled b. For j = 0 all leaves are labelled v.

$$t^0_0 = v$$

$$t^0_1 = b$$

$$t^{i+1}_j = a(t^i_j, t^i_0) \qquad 0 \le j \le 2^i$$

$$t^{i+1}_j = a(t^i_0, t^i_{j-2^i}) \qquad 2^i < j \le 2^{i+1}$$

Define the pattern forest $F_n = \{t^n_i \mid 1 \le i \le 2^n\}$. Clearly the size of F_n is $O(2^n)$. Furthermore, it is easy to see that $t^n_i \sim t^n_j$ for distinct values of i and j. Consider sets Q of integers between 1 and 2^n, and define for each such Q a balanced binary tree t_Q of height n with all interior nodes labelled a, and such that the i-th leaf from the left is labelled b if i is in Q, c otherwise. Surely t^n_i matches t_Q at the root iff $i \in Q$. There are 2^{2^n} such sets Q, thus there must be at least as many different match sets. ■

As a consequence, a preprocessing algorithm based on computing tables indexed by possible match sets, as would be needed to drive Algorithm M, must be impractical, in certain cases. Since independence among (sub)trees in pattern forests is responsible for a possible exponential growth of the number of match sets, it is useful to have an intuitive understanding of this relation. The following proposition provides a basis for this.

Proposition 2.5 Two trees t and t' are independent only if t contains disjoint subtrees t_1 and t_2, and t' contains disjoint subtrees t'_1 and t'_2 such that $t_1 > t'_1$ and $t'_2 > t_2$.

Proof Since v and constant symbols cannot be independent of other trees, we may assume

$$t = a(t_1, .., t_k)$$
$$t' = a(t'_1, .., t'_k)$$

We prove the proposition by induction on the height of t.

If the height of t is 1, then the t_i all have height 0, hence are either v or constants. For all $i \leq k$, therefore, $t_i > t'_i$ or $t'_i > t_i$, otherwise t and t' would be inconsistent. Since neither $t > t'$ nor $t' > t$, there must be corresponding pairs t_i, t'_i and t_j, t'_j satisfying the proposition.

Assume then that the proposition holds for all trees t of height less than h, and let t be of height h. If, for some i, $t_i \| t'_i$, then $t \| t'$, contrary to assumption. If, for some i, $t_i \sim t'_i$, then the proposition follows from the induction hypothesis. Otherwise the argument of the induction basis completes the proof. ■

The mutual subsumption, in opposite direction, of disjoint subtree pairs is not a sufficient condition for independence, since it does not rule out possible inconsistency: $a(b,v,c) \| a(v,b,d)$, yet there are disjoint subtree pairs satisfying the only if condition of Proposition 2.5.

3. Simple Pattern Forests

Because of the exponential growth of the number of match sets for certain pattern forests, we wish to restrict patterns when generating tables to drive Algorithm M of Section 2. Theorem 2.3 suggests disallowing independence among pattern (sub)trees. Although this might seem a drastic step, it has not seriously hindered axiomatizing LISP, Lucid, or the Combinator Calculus, for which interpreters have been generated using these techniques, [H&O 79]. We therefore make the following

Definition A pattern forest F is simple if it contains no independent (sub)trees.

For simple forests, the independence graph has no edges, hence, by Theorem 2.3, the number of different match sets is at most the size of the forest. Given a pattern forest F, define immediate subsumption, $>_i$, as follows: $t >_i t'$ iff $t > t'$ and there is no other (sub)tree t" in F such that $t > t''$ and $t'' > t'$. Note that immediate subsumption is the transitive reduction of subsumption on the set of all (sub)trees in F.

The immediate subsumption graph G_S of the pattern forest F is as follows. The vertices of G_S are the distinct (sub)trees in F. There is a (directed) edge from t to t' iff $t >_i t'$. In general, G_S is a directed acyclic graph, with a single leaf which is the symbol v.

Lemma 3.1 The immediate subsumption graph G_S of a simple forest F is an inverted tree, with v as root.

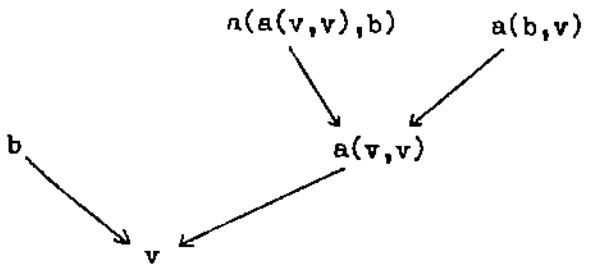
Proof Let t, t' and t" be different (sub)trees in F, and assume that t subsumes both t' and t", but neither $t' > t''$ nor $t'' > t'$. Since t subsumes both trees, $t' \| t''$ is impossible (Proposition 2.1 d), hence t' and t" should be independent. But then F cannot be simple. Hence either $t' > t''$ or $t'' > t'$. ∎

Note that, for simple forests F, the base set M_0 of any match set must be a singleton. We thus obtain, using Lemma 3.1 and Proposition 2.2,

Theorem 3.2 Let F be a simple forest, and M any match set for F. Let the base set of M be $\{t\}$. Then M is the set of trees encountered on the path from t to v in G_S, including t.

G_S thus provides at once the set of all possible match sets, and their structure.

Example 3.1 The pattern forest of Example 2.2 is simple. Its immediate subsumption graph is



From G_S we obtain as the possible match sets

$\{v\}$
$\{b, v\}$
$\{a(v,v), v\}$
$\{a(a(v,v),b), a(v,v), v\}$
$\{a(b,v), a(v,v), v\}$

There is a connection between the immediate subsumption graph G_S and the "failure function" f used in the string matching algorithms in [KMP 77, A&C 75]. This connection is observed as follows: Given a string pattern $a_1a_2...a_m$, consider it as a nonbranching tree $a_m(a_{m-1}(... a_1(v) ..))$. The variable symbol v is added as leaf so as to assign a consistent arity of 1 to each of the a_i, and to permit the pattern to "slide" in the subject. Matching in a string $b_1...b_n$ is considered to be matching in the nonbranching tree $b_n(... b_1(c) ..)$, where

c is a fresh symbol of arity 0. Translated to a tree matching problem in this way, G_S is precisely the graph of the failure function f constructed for the original string problem by the algorithms in [KMP 77 and A&C 75]. For this note that a subtree corresponds to a pattern prefix, and that t subsumes t' iff t' is a pattern prefix which matches, as suffix, in the pattern (prefix) t. Hence $t >_i t'$ iff t' is the longest proper prefix which matches, as suffix, in the (prefix) t, which is just the definition of f. Note also, that because of Proposition 2.5 pattern forests derived from string patterns must be simple, because nonbranching trees cannot have disjoint subtrees. Hence there is no counterpart in string matching to the exponential explosion of match sets, which can occur for nonsimple forests in the case of tree matching.

4. Table Generation for Simple Forests

We can construct the tables needed to drive Algorithm M in two steps for a simple pattern forest F: Compute the immediate subsumption tree G_S. From it, we have a representation of all possible match sets. Then construct a table for each alphabet symbol, filling in entries using the information provided by G_S.

The computation of G_S by Algorithm A below is based on these observations. If t subsumes t', in particular, if $t >_i t'$, then the height of t cannot be less than the height of t'. Furthermore, if $t >_i t'$ and $t = a(t_1, .., t_k)$, then either $t' = v$, or $t' = a(t'_1, .., t'_k)$ such that, for $1 \leq i \leq k$, $t_i > t'_i$ or $t_i = t'_i$. However, in the latter case, at least one of the pairs t_i, t'_i will consist of unequal trees.

Algorithm A Compute G_S for simple pattern forest F.

Input: Simple pattern forest F.

Output: Subsumption tree G_S for F.

Note: $t >_i t'$ is denoted by $f(t) = t'$.
The set of all (sub)trees in F is denoted by T(F).

Method

1. Order all trees in T(F) by their height.
2. For each $t \neq v$ in T(F) of height 0 enter $f(t) = v$.
3. For h := 1 to maximum height in T(F) do
4. For each $t = a(t_1, .., t_k)$ in T(F) of height h do begin
5. s := v;
6. For i := 1 to k do begin
7. t' := $f(t_i)$;
8. Examine the trees $t'' = a(.., t', ..)$ of height $h'' \leq h$ and with t' as i-th immediate subtree ordered by decreasing height h'':
9. If none of the t'' is subsumed by t and if $t' \neq v$ then set t' := f(t') and repeat Step 8.
10. For each t'' subsumed by t found in Steps 8 and 9 and of maximal height h'' do
11. if $t'' > s$ then s := t'';

12. end; (of loop in i)
13. Enter f(t) = s;
14. end; (of loop for each t)

Steps 5 through 13 compute the tree $\bar{s}$ which is immediately subsumed by t. If $\bar{s} \neq v$, then $\bar{s} = a(f^{i1}(t_1), .., f^{ik}(t_k))$ where i1 + i2 + ... + ik is minimum but greater zero. Since we cannot predict which (of the) ij > 0, we explore, separately for each value of i, the individual subtree positions in Steps 6 through 12. Suppose then we find a tree $t'' = a(t''_1, .., t''_k)$ where $t''_i = t'$ in Step 8. This tree is a candidate for $\bar{s}$ provided that $t > t''$. We test this by testing, for each pair t_i, t''_i, if $t_i > t''_i$, or $t_i = t''_i$. Since the trees t_i and t''_i are of height strictly less than h, we do this by examining, in the constructed portion of G_S, the path t_i, $f(t_i)$, .., v, from t_i to the root of G_S. If t''_i is encountered, then $t_i > t''_i$.

Now consider that there might be several candidates for $\bar{s}$. By Lemma 3.1 $\bar{s}$ subsumes all other trees subsumed by t, hence the test of Step 10. Finally, for any tree t'' such that t'' is subsumed by t, the height of t'' cannot exceed the height of $\bar{s}$. Thus we process all candidates by decreasing height in Steps 8 and 9. For this, we build <u>indexing lists</u>: For each tree t there are k·s lists where k is the maximum arity of the alphabet, and s is its size. If t is the i-th immediate subtree of some tree t' with root a, then t' is in the ⟨a,i⟩ indexing list of t. The lists contain trees ordered by decreasing height, and are extended each time h is stepped, between Steps 3 and 4, by entering each tree t of height h into the appropriate lists in stack fashion. Since it is possible that a tree and its immediately subsumed tree are of the same height, we must enter all trees of height h into these lists before beginning the computation of Steps 4 through 14.

For deriving the complexity of Algorithm A we use the following:

m - size of the pattern format F

d - height of G_S

s - size of the alphabet Σ

k - highest occurring arity in Σ

Sorting T(F), which contains m trees, can be done in O(m) steps using a bucket sort. Steps 5 - 13 are executed for each of the O(m) trees in T(F) of height greater than 0. In these, there are k passes, during each of which at most O(m) trees t'' are considered in Steps 8 and 9. Testing $t > t''$ and $t'' > s$ requires no more than k·d steps, if done by traversing the existing part of G_S. Choosing trees from the indexing lists can be done in constant time, since the lists are maintained sorted by decreasing height. Thus, each of the k passes (Steps 7 - 11) requires at most O(m·k·d) steps. Thus, for each t considered, Steps 5 - 13 take at most $O(m \cdot k^2 \cdot d)$ steps. In addition to this computation, the indexing lists have to be maintained. This involves initializing k·s lists for each t, and adding a tree t to at most k lists. Thus, the algorithm requires a total of at most $O(m \cdot k \cdot s + m^2 \cdot k^2 \cdot d)$ steps. Since k and s depend only on the alphabet, we have an $O(m^2 \cdot d)$ algorithm for fixed alphabets.

Since G_S is a tree, additional space requirements can only arise with the indexing lists. As each tree t can be in at most k different such lists, however, the algorithm requires O(m) space. In summary,

Theorem 4.1 Algorithm A reqires at most $O(m^2 \cdot d)$ steps and O(m) space.

Since F must be simple, we should modify Algorithm A suitably to test this. This is done by testing, in Step 9, both

(a) For $1 \leq i \leq k$: $t_i > t''_i$ or $t_i = t''_i$

(b) For $1 \leq i \leq k$: $t_i > t''_i$ or $t_i = t''_i$ or $t''_i > t_i$

If (b) but not (a) holds for any t'' of maximal height, then F is not simple. See also Proposition 2.5. It can be proved that this addition is sufficient to test simplicity of F.

In Section 3 we observed the connection between G_S and the failure function of string pattern matching. There is also an (indirect) connection between Algorithm A, and the computation of f proposed in [KMP 77, A&C 75]: Note that, for nonbranching trees, the loop of Step 6 is not needed, since k = 1. Furthermore, in Step 8, exactly one tree t'' exists with immediate subtree t', which is, at the same time, subsumed by t. Steps 8 and 9 thus become, essentially,

8' while there is no tree a(t') and t' ≠ v do

9' t' := f(t');

With these considerations, Algorithm A may be considered a generalization of the string failure function algorithms.

Once G_S has been computed, we have, by Theorem 3.2, computed all match sets. With it, we can generate tables as follows.

Algorithm B Table Generation

Input: G_S of simple pattern forest F

Output: Tables to drive Algorithm B

Method

1. Traverse G_S in postorder. For each tree $t = a(t_1, .., t_k)$ encountered do
2. Assign t to each entry of the table for a which is not yet assigned and which is indexed by the tuples $\langle t'_1, .., t'_k \rangle$, where, for $1 \leq i \leq k$, $t'_i > t_i$ or $t'_i = t_i$.
3. Enter v into the remaining unassigned entries of each table.

The algorithm is best understood by considering some tree with root a at whose i-th son we have found the (partial) matches forming the set with base $\{t'_i\}$. Since $t'_i > t_i$ or $t'_i = t_i$, the tree $a(t_1, .., t_k)$ must match at the root. Thus Step 2 of Algorithm B assigns to each table entry a member of the correct match set.

Observe now that we traverse G_S in postorder. Thus, if a tree t' could be assigned to an entry already assigned t, $t > t'$ must hold, therefore we assign to each entry indexed by $\langle t'_1, \ldots, t'_k \rangle$ (of the table for a) the base set tree of the match set which applies to any node labelled a at whose i-th son we have matched all trees forming the match set with base $\{t'_i\}$.

The table for a symbol $a \in \Sigma$ of arity k has m^k entries, where m is the size of the pattern forest F, as above. Thus Algorithm B constructs no more than $m^k \cdot s$ entries, where k is the highest arity in Σ and s is the alphabet size. Since the overlap in assigning an entry cannot exceed the size of the largest match set, i.e. the height of G_S, at most $m^k \cdot s \cdot d$ assignments are attempted by Step 2. Note that the tuples range precisely over the subtrees of G_S rooted in the t_i, hence we can find all tuples easily. We obtain, in summary,

Theorem 4.2 Algorithm B requires at most $O(m^k \cdot d)$ steps and $O(m^k)$ space.

Example 4.1 The tables generated from G_S of Example 3.1 are

For a: left subtree \ right subtree matches	v	b	a(v,b)	a(v,v)	a(a(v,v),b)
v	a(v,v)	a(v,v)	a(v,v)	a(v,v)	a(v,v)
b	a(b,v)	a(b,v)	a(b,v)	a(b,v)	a(b,v)
a(v,v)	a(v,v)	a(a(v,v),b)	a(v,v)	a(v,v)	a(v,v)
a(b,v)	a(v,v)	a(a(v,v),b)	a(v,v)	a(v,v)	a(v,v)
a(a(v,v),b)	a(v,v)	a(a(v,v),b)	a(v,v)	a(v,v)	a(v,v)

The table for b has just one entry, which is b.

Clearly Algorithm B constitutes the bottleneck of the preprocessing, both in space and time requirements. Often the situation can be improved by introducing pairing functions, thereby reducing k to 2. There are, however, cases in which pairing may destroy the simplicity of the forest, introducing independence:

Example 4.2 Consider the pattern forest formed by the trees $t_1 = a(b,v,c)$, $t_2 = a(v,b,d)$, $t_3 = a(e,c,v)$. All (sub)trees other than v are pairwise inconsistent, thus the forest is simple. Introduction of a pairing function p, no matter which subtrees of a are paired, will introduce independence. For example, pairing the first and second subtrees results in a new forest

$$\{a'(p(b,v),c),\ a'(p(v,b),d),\ a'(p(e,c),v)\}$$

with independent subtrees p(b,v) and p(v,b).

There is a different approach to speeding up the preprocessing. Recall that G_S generalizes the failure function f of string matching. If a different matching algorithm M' could be designed which uses G_S to do the matching, then the expensive Algorithm B could be bypassed. Indeed, we may adapt Steps 5 - 13 of Algorithm A to do the matching. Properly done, finding all occurences of patterns forming a simple forest of size m in a subject tree of size n would then require O(n·m·d) steps, where d is the height of G_S. Unfortunately, the worst case here is slower than the performance of the naive algorithm, due to the time bound on the computation of $t >_i t'$ (of O(m·d)). If this computation (Steps 5-13) could be organized faster, a competitive matching algorithm would be possible which does not use large tables and requires less preprocessing. We develop such an algorithm in Section 5.

5. A Different Approach

The reason why Algorithm A is quadratic in the size of the forest is that Steps 8 and 9 also consider trees $t'' = a(.., t', ..)$ which are found to be inconsistent with t. If we could exclude inconsistent trees, then Algorithm A could be sped up. The approach we develop now requires alphabets of function symbols which are at most binary.

Consider a tree $t = a(t', t'')$. Any tree subsumed by t other than v is of the form $a(f^i(t'), f^j(t''))$, $i+j>0$. Here f denotes immediate subsumption, and the iterates f^i of f have their usual meaning. If F is a simple pattern forest, then all occurring trees of this form may be linearly ordered by subsumption (Lemma 3.1), and there will be at most d such trees, where d is the height of G_S. What is needed is a good data structure for finding trees of this form rapidly. We use sets $S(a,t')$, a in Σ_2, t' in F, where $S(a,t')$ contains the pair $\langle t'', t\rangle$ iff F contains a tree $t = a(t',t'')$. We can then search for all trees of the required form by testing, for $i+j>0$, if a pair $\langle f^j(t''), *\rangle$ is in the set $S(a,f^i(t'))$. Provided the test can be done in constant time, we can find all trees of the required form in $O(d^2)$ time.

A set representation which permits membership test in constant time is the characteristic vector: The set $S(a,t')$ is represented as a vector indexed by trees t''. Note that since a, t', and t'' completely determine the tree, we could represent vector entries as bits. That is, a pair $\langle t'', t\rangle$ which is member of $S(a,t')$ is an entry in the vector for $S(a,t')$ at position t''. There are at most $s \cdot m$ such sets, where m is the size of the pattern forest, and s is the alphabet size, each occupying a vector of size m. In order to avoid

an m^2 time requirement for initializing the vectors, we use constant time initialization (see [AHU 74], Ex. 2.12). Note that a tree a(t',t") is entered into just one set. We can therefore reduce space requirements by delaying the allocation and initialization of the vectors until needed by a set member to be entered. There will be a total of m entries into all allocated vectors combined.

Note that for trees a(t') we could simply add a special set S'(a,t') consisting of at most one element. The search for an immediately subsumed tree is then even simpler, and in fact like the computation of f in [KMP 77]. The required additional steps are routine and have not been indicated in Algorithm C.

Algorithm C Computation of G_S for simple binary forest F

Input: Simple pattern forest F over an alphabet of highest arity 2
Output: Immediate subsumption tree G_S
Method

1. Order all trees t in T(F) by height and mark each set S(a,t) "not allocated"
2. For all t ≠ v in T(F) of height 0 enter f(t) = v
3. For h := 1 to maximum height in forest do begin
4. For each t = a(t',t") in T(F) of height h do
5. Enter t in S(a,t')[t"] (if necessary, allocate and initialize a vector of size m for the set S(a,t') first)
6. For each t = a(t',t") in T(F) of height h do begin
7. Search through sets $S(a,f^i(t'))$, i = 0, 1, ..., for an entry at $f^j(t")$, j = 0, 1, ... and stop with the first entry s found, if any exists.
8. If an entry s has been found in Step 7, then enter f(t) = s, otherwise enter f(t) = v.
9. end;
10. end.

As explained above, the double loop in Step 7 requires $O(d^2)$ membership tests, where d is the height of $G_{..}$. From this, and the other observations above, follows

Theorem 4.1 Algorithm C requires at most $O(m \cdot d^2)$ steps and $O(m^2)$ space, where m is the size of the simple pattern forest processed, and d is the height of the tree G_S.

Note that the two loops of Steps 4-5 and 6-9 of Algorithm C cannot be combined, since the tree immediately subsumed by some t may be of the same height as t.

We may use Algorithm C for alphabets of higher degree after introducing pairing functions. For the reasons illustrated by Example 4.2 certain simple forests over higher degree alphabets become nonsimple through this transformation and cannot be processed, therefore.

We may use Steps 7 and 8 in designing a matching algorithm which does not use the tables as generated by Algorithm B. This new matching algorithm processes a subject tree in which to match from the leaves up, just as Algorithm M does. When processing a node labelled a at whose left and right son we have matched sets with base set trees t' and t", respectively, we execute first Step 7 of Algorithm C. Step 8 is then performed, but altered such that s (or v) is assigned to the node presently processed, rather than assigning f. The resulting algorithm finds all matches in $O(n \cdot d^2)$ time in a subject of size n.

6. Conclusions

We have seen in Sections 4 and 5 how the special properties of simple pattern forests can lead to efficient solutions of the matching problem for patterns of this kind. For nonsimple forests, the immediate subsumption graph G_S need not be an (inverted) tree. It is not difficult to extend Algorithm A to compute G_S in the general case within the same time bound. This might become a useful step for preprocessing nonsimple pattern forests, since we can prove a generalization of Theorem 5.2: If M is a match set with base set M_0, then it consists of the trees encountered on all paths from the trees $t \in M_0$ to v in G_S. G_S does not, however, provide sufficient information on which base sets are possible.

Because of Theorem 2.4, further research is needed to discover if more extensive pattern classes than simple forests exist, which still possess efficient matching algorithms. Until such classes are found, the naive algorithm for tree pattern matching remains a safe and reasonable solution for the matching problem for nonsimple pattern forests.

References

[A&C 75] A.V.Aho and M.J.Corasick
Efficient String Matching: An Aid to Bibliographic Search
Comm. ACM 18:6 (1975) 333-343

[AHU 74] A.V.Aho, J.E.Hopcroft, and J.D.Ullman
The Design and Analysis of Computer Algorithms
Addison-Wesley, Reading Mass., 1974

[Bax 76] L.D.Baxter
The Complexity of Unification
Ph.D. Diss., Dept. of Comp. Science, Univ. of Waterloo,
Waterloo, Ont., Canada, 1976

[BM 77] R.S.Boyer and J.S.Moore
A Fast String Searching Algorithm
Comm. ACM 20:10 (1977) 762-772

[Col 71] G.Collins
The SAC-1 System: An Introduction and Survey
Proc. 2nd ACM Conf. on Symb. and Algebr. Manipulation,
Los Angeles, 1971, 144-152

[DST 76] P.J.Downey, B.Samet, and R.Sethi
Off-Line and On-Line Algorithms For Reducing Equalities
Proc. 5th Annl ACM Symp. on Princ. of Progr. Lang.,
Tucson, Ariz., 1978, 158-170

[GHH 76] J.Guttag, E.Horowitz, and D.Musser
Abstract Data Types and Software Validation
Inf. Sci. Institut. Rept. ISI/RR-76-48, University of Southern
California, 1976

[HO 79] C.M.Hoffmann and M.J.O'Donnell
An Interpreter Generator Using Tree Pattern Matching
Proc. 6th Annl ACM Symp. on Princ. of Progr. Lang.,
San Antonio, Texas, 1979

[K&B 70] D.Knuth and P.Bendix
Simple Word Problems in Universal Algebras
in Computational Problems in Abstract Algebra, J. Leech, ed.,
Pergamon Press, Oxford, 1970, 263-297

[KMP 77] D.Knuth, J.Morris, and V.Pratt
Fast Pattern Matching in Strings
SIAM J. on Computing 6:2 (1977) 323-350

[N&O 78] G.Nelson and D.Oppen
Fast Decision Algorithms Based on Congruence Closure
Stanford AI Memo AIM-309, Stanford University, 1978

[O'D 77] M.J.O'Donnell
Computing in Systems Described by Equations
Springer Lecture Notes in Comp. Science #58, Springer Verlag, New York, 1977

[P&W 76] M.S.Paterson and M.Wegman
Linear Unification
Proc. 8th ACM Symp. on Thy. of Computing, Hershey, Penn., 1976, 181-186

[Rob 65] J.A.Robinson
A Machine Oriented Logic Based on Resolution
J. of ACM 12:1 (1965) 23-41

[Ros 73] B.Rosen
Tree Manipulating Systems and Church-Rosser Theorems
J. of ACM 20:1 (1973) 160-187

[Sho 78] R.E.Shostak
An Algorithm for Reasoning About Equality
Comm. ACM 21:7 (1978) 583-585

[Sta 77] G.Stafford
Structure of the Eh Compiler
M.S. Thesis, Dept. of Comp. Science, Univ. of Waterloo, Waterloo, Ont., Canada, 1977